ISRN SICS-T--91/8--SE

Implementing a Module System for SICStus Prolog

by
Stefan Andersson

SICS technical report
T91:08
ISSN 1100-3154

Table of Contents

CHAPTER 1

Modularity in Prolog

1.1 Why Modules?

A prolog predicate is identified by its name and arity. In a non-modular Prolog, these must be unique in the whole system. It is obvious that this is a major disadvantage, especially when large applications are developed and more than one programmer are involved. A module system makes it possible to partition the programs into modules which may correspond to functional entities of the application. Each module has its own independent name space and an interface to other modules. The interface consists of a set of predicates which are declared as public and may be imported by other modules.

1.2 Module Concepts

There are lots of things that might be said on the subject of module systems for Prolog. Since this report concerns only the implementation of an almost complete specification, I will not go too deeply into these matters.

Apparently, there are basically two different module concepts. In the first one, which we call the *name based concept*, both data and procedures are local to a module. That implies for example that if the same text string is entered into two different modules as an atom, they become different atoms and cannot be unified with each other. The other concept is the *procedure based*, which is the one we are using for the ISP. Here the data are all global, only the procedures are local to the modules.

There are some pros and cons associated with both concepts. Prolog makes no sharp distinction between program and data; a term might be referring to a procedure by means of built in meta-predicates. While this is no problem for the name based concept, in the procedure based concept each term used as a procedure reference must include a module name to distinguish between procedures with the same name in different modules.

A clarifying example: The goal `assert(t(X))` appears in a procedure `p`, (`assert/1` is a meta predicate). In a name based system, the term `t(X)` becomes local to the module of `p` at load time and when the assert is executed, the predicate `t/1` will automatically belong to the same module.

In the procedure based system, `t(X)` is a global term and therefore assert must be supplied with a module specification as well as the term to make a predicate of it. We shall later see how this is handled in our module system.

The name based concept on the other hand has a number of drawbacks. Some of these are subjective:

- It takes a more complex structure for storing the atoms.
- You seem to need a lot of declarations.
- You have to be very careful what you type in at the toplevel so you don't define atoms that really should be imported.
- You can't, for example, import the predicates foo/1 and foo/2 from different modules because it is actually foo you import.

1.3 Module Visibility

Another property of a module system is whether it is *flat* or *hierarchical.* In a flat module system, all the modules are visible to each other, while in the hierarchical ditto, it is possible to define submodules which are only visible inside the super-module where they are defined. An analysis of the advantages and disadvantages of these properties falls outside the scope for this work. The ISP module system is flat.

CHAPTER 2

The Module System

2.1 Introduction

The choice of module system for the ISP has fallen upon a Quintus compatible one. There are several reasons for this:

- *Quintus Prolog* is widely spread and has become something of a de facto standard.
- The Quintus module system is fairly simple, understandable and easy to use.
- It is possible to implement without making too many radical changes to the Sicstus and without any significant degrading of performance.

The specification contains the Quintus compatible part which is based on information from the Quintus manual. Further on, the module system must adapt the Sicstus specific features, like delayed goals and compiled files.

2.2 Basic Concepts

The module system is procedure based. This means that it is only the predicates that are local to a module; all data, functors and atoms, are global. The module system is flat, not hierarchical, so all modules are visible to one another.

All predicates in the prolog system, both built in and user defined, belong to some module. A predicate is generally only visible in the module where it is defined. However a predicate may be *imported* by another module. It is thereby made visible in that module too. Predicates declared as public in a *module declaration* (see below) are *exported.* Normally only exported predicates may be imported.

There are two predefined modules, the *prolog* module and the *user* module. All the built in predicates reside in the prolog module. The exported built in predicates are automatically imported into each other module as it is created.

The user module is the default module where all user defined predicates go if there is no module specification. In this way the system may be used without the user being aware of the module system at all.

2.3 Module Specification

The module of a predicate call may be specified by prefixing the goal with the module name and the colon operator:

```
?- foo:bar(X).
```

When there is no explicit module specification, all goals appearing in files refer to the *source module*. Goals issued at the top level refer to the *type-in module*. The type-in module is by default the user module but may be changed by the directive:

```
?- module(ModuleName).
```

The source module of a file is either the module specified in a module declaration or, for non module files, the type-in module.

2.4 Defining Modules

A module may be defined by putting a *module declaration* first in a source file.

```
:- module(ModuleName, PublicPredicateList).
```

When the file is loaded, all predicates in the file go to the stated module and the predicates of the `PublicPredicateList` are exported. When a module declaration is processed, all existing predicates in the module are erased before the new ones are loaded. A file which contains a module declaration is henceforth called a *module file*. There may be only one module declaration per file and it must appear first in the file.

A module can also be defined dynamically by asserting or loading predicates to it:

```
?- assert(m:p(x)).
```

creates the module `m`, if it doesn't exist already, and asserts `p(x)` to it.

```
?- compile(m:f).
```

creates the module `m` and loads `f` into `m`.

Dynamically created modules have no exported predicates.

2.5 Importation

When a module file is loaded, predicates are imported by the receiving module. `consult/1`, `compile/1`, `ensure_loaded/1` and `load/1` import all the exported predicates with the exception that if any of these predicates are previously imported from the same module, no importation at all takes place. There are two other built in predicates for loading module files that have the same loading characteristics as `ensure_loaded/1` but differ in importation.

`use_module(+Files)`
: Always imports all exported predicates of `Files`.

`use_module(+File, +PredicateList)`
: Always imports the predicates in `PredicateList`. If any of these are not exported, a warning is issued.

Clashes with already existing predicates, local or imported from other modules, are handled in two different ways: If the receiving module is the user module, the user is asked for redefinition of the predicate. For other receiving modules, a warning is issued and the importation is canceled. The binding of an imported predicate remains, even if the origin is reloaded or deleted. However `abolish/1-2` breaks up the importation bindings.

When a module file is reloaded, a check is made that the predicates of this file, imported by other modules, are still in the public list. If that is not the case, a warning is issued. Note that an imported predicate may be reexported.

2.6 Meta Expansion

Meta predicates and other predicates that need module specification must have their arguments *module-name-expanded.* Goals issued at toplevel and goals appearing as directives are expanded prior to execution while goals in the bodies of clauses are expanded at compile time. The expansion is made by prefixing the argument with `Module:`, where `Module` is the source module or the type-in module. An argument is not expanded if:

1. It has a module prefix already.
2. It is a variable which appears in a meta expandable position in the head of the clause.

Some clarifying examples:

`p/1` and `q/1` are meta predicates while `r/1` is not. The clause `r(X) :- p(X)` will be transformed to `r(X) :- p(M:X)` while `q(X) :- p(X)` will not, due to item 2.

```
?- m:assert(f(1)).
```

`assert/1` is called in the module `m` but to ensure that `f(1)` is asserted into `m`, the goal must be transformed to `m:assert(m:f(1))` before execution.

2.7 Meta Declarations

Predicates which need meta expansion are defined in the following way:

```
:- meta_predicate p(:, +).
```

This means that the first argument of `p/2` shall be expanded. Instead of the ':', an integer also means expansion while anything else, like `+`, `-` or `?` for example, means no expansion. A number of built in predicates have predefined meta declarations as follows:

```
:- meta_predicate
        predicate_property(:,?), current_predicate(?,:), call(:),
        call_residue(:,?), freeze(:), freeze(?,:), frozen(?,:),
        cavalier_freeze(?,:),

        assert(:), assert(:,-), asserta(:), asserta(:,-),
        assertz(:), assertz(:,-), retract(:), retractall(:),
        abolish(:), abolish(:,+), clause(:,?), clause(:,?,?),

        consult(:), reconsult(:), compile(:), load(:),
        ensure_loaded(:), use_module(:), use_module(:,+),
        fcompile(:),

        load_foreign_files(:, +), incore(:),
        findall(?,:,?), bagof(?,:,?), setof(?,:,?),
        (? ^ :), [:|+], spy(:), nospy(:),
        (:,:),
        (:;:), (: -> :), (\+ :), if(:,:,:),
        listing(:), portray_clause(:),

        call_residue(:,?,?), save_instances(?,:,?,?).
```

Note that the control structure predicates (',', ';' etc) are meta declared. The reason for this is that when directives like

```
?- m:(g(X);h(X)).
```

are given, the module prefix has to be distributed to `g(X)` and `h(X)` in order to make the calls in the right module. This is accomplished by meta expansion.

2.8 Modules and File-to-File Compilation

Since the module name expansion takes place at compile time, the module to which the file is to be loaded must be known when compiling to object files. This is no problem for module files because the module name is picked from the module declaration. When non-module files are compiled, the file name may be prefixed with the module name which is to be used for expansion.

```
:- fcompile(Module:Files).
```

If an object file is loaded to a different module from which it was compiled for, a warning is issued.

There is a limitation to the use of meta predicates for object file compilation: The only meta declarations visible to the compiled file are the ones in the same file and the built in ones. Meta predicates which are used in the file but defined in other files (i.e. imported) are not expanded because the meta declarations are not visible.

2.9 Modules and the Emacs Interface

When code is loaded through the Emacs interface, the module where the loading goes to is the same as the module to which the file was originally loaded. If the file has not been loaded before, it goes to the type-in module. If the type-in module is not the user module, the user is asked if the loading should take place. When a piece of a module file which contains the module declaration is loaded, the behavior is the same as when doing a compile/consult except that no importation takes place. Note that when a module declaration is processed, all previous definitions in the module are erased. It will normally not be meaningful to include the module declaration except when the whole file is loaded.

2.10 Built in Predicates for Module Handling

`+Module:+Goal`
: `Goal` is meta expanded and issued in `Module`.

`use_module(+Files)`
: Loads `Files` like `ensure_loaded/1` but always imports all exported predicates.

`use_module(+File, +PredicateList)`
: Loads `File` like `ensure_loaded/1` but always imports the predicates in `PredicateList`.

`current_module(?Module)`
: `Module` is a module defined in the system. It can be used to backtrack through all modules presently in the system.

`current_module(?Module, ?File)`
: `Module` is the module defined in `File`.

`module(+Module)`
: The type-in module is set to `Module`.

2.11 Module Extensions to Built in Predicates

A number of built in predicates take a module prefix to one of their arguments. If the module prefix is omitted, the type-in module is used. The following table describes the extended predicates in their prefixed forms. The meaning of the module specification is explained when it is not obvious.

```
consult(+Module:+Files)
reconsult(+Module:+Files)
[+Module:+File|+Files]
compile(+Module:+Files)
load(+Module:+Files)
ensure_loaded(+Module:+Files)
```
: The files are loaded to `Module` instead of the type-in module.

`fcompile(+Module:+Files)`
: `Module` is used for meta expansion in `Files`. `freeze(+Goal)`

```
freeze(?X, +Goal)
frozen(-Var, ?Goal)
call(+Term)
```

`incore(+Term)`
`call_residue(+Goal, ?Vars)`
`setof(?Template,+Goal, ?Set)`
`bagof(?Template,+Goal, ?Bag)`
`X^Goal`
`findall(?Template, +Goal, ?Bag)`
In these predicates, `Goal/Term` may be module prefixed to indicate into which module the goals are called.

`listing(?Spec)`
`+Spec` is as described under `spy/1` with the exception that `?Spec` may contain variables. Example:

```
?- listing(user:_).
```

Lists all interpreted predicates in the module `user`.

`current_predicate(?Name, ?Module:?Head)`
`Head` is the most general goal of a predicate defined in `Module`.

`predicate_property(?Module:?Head, ?Property)`
`Head` is the most general goal of a predicate visible in `Module`. The set of properties are extended with `exported` which means that the predicate is public, `imported_from(ModuleFrom)` telling where the predicate is imported from and `meta_predicate`. The specification for meta expansion given by a meta declaration is reflected by the property `meta_predicate(Spec)`.

`assert(+Module:+Clause)`
`assert(+Module:+Clause, -Ref)`
`asserta(+Module:+Clause)`
`asserta(+Module:+Clause, -Ref)`
`assertz(+Module:+Clause)`
`clause(+Module:+Head, ?Body)`
`clause(+Module:+Head, ?Body, ?Ref)`
`clause(?Module:?Head, ?Body, +Ref)`
`retract(+Module:+Clause)`
`retractall(+Module:+Head)`
The affected predicate is visible in `Module`.

`abolish(+Spec)`
`abolish(+Module:+Name, +Arity)`
The affected predicate must be defined in `Module`. `+Spec` is described under `spy/1`.

`load_foreign_files(+Module:+ObjectFiles,+Libraries)`
: The user defined predicates `foreign_files/2` and `foreign/2-3` are sought in `Module` and the predicates defined are placed in `Module`.

`spy +Spec`
`nospy +Spec`
: `+Spec` can have the forms: `Name`, `Name/Arity`, `Name/Low-High` with or without module prefix or a list of such, which may also have a module prefix. Examples:

```
?- spy [user:p/1, m:q/2].
?- spy m:[p/1, q/1].
```

2.12 Ensure Loaded

Although this predicate is not a part of the module system, it has been implemented at the same time. To make sure a program is loaded from a set of files, use the built-in predicate:

```
|?- ensure_loaded(Files).
```

where Files is either a single filename or a list of filenames. Whenever a file is loaded, the filename is recorded in the system (for object files, the corresponding source filename). When `ensure_loaded/1` is issued, the system checks if the file is already loaded. If that is the case, no loading is done. If the file has not been loaded, the system tries to find a source file (with or without a '.pl') and an object file (with a '.ql'). In case both are present, the object file is chosen if it is newer than the source file, and is loaded as with `load/1`. Otherwise the source file is compiled or consulted. The file is consulted if the goal appears as a directive in a file which is consulted, otherwise it is compiled.

2.13 Delayed Goals

The delay facility is modified so that goals are frozen with module prefixes. This is most notable for the `frozen(-Var, ?Goal)` predicate which returns goals on the form `Module:Goal`.

CHAPTER 3

Implementation

3.1 Introduction

This chapter describes in detail the implementation of the module system. It requires good knowledge of the Sicstus and access to the source code to be of any greater use.

3.2 Data Structures

As mentioned above the predicates definition records are looked up in hash tables. For the module system a new hash table is introduced where module records are looked up. This module record in turn contains a pointer to the hash table of predicates belonging to the module. The module hash table uses the same hash functions as the predicate hash tables.

```
struct mod_def {
    TAGGED name;
    BOOL empty;
    struct definition *public_lst;
    struct sw_on_key *preds;
};
```

The system module, which has the name `prolog`, although it is contained in the modules hash table as any other module, has a special global pointer to it. There is also a module pointer which points out the present type-in module. As the type-in module and the current source module are in fact equal, the `typein_mod` is used during loading/compilation as pointer to the source module.

```
struct mod_def *prolog_module;
struct mod_def *typein_mod;
```

Furthermore the module record contains the module name, which is an atom. The rest of the fields are explained later on.

The predicates definition record is extended as follows. The module field is a pointer to the module where the predicate is defined, which is not necessarily the same as where the definition record is looked up. The use of the fields `module`, `exported_to` and `imported_from` are explained in depth under importation.

```
struct definition {
        short enter_instr;          /* see predtyp.h */
        short arity;
        struct mod_def *module;     /* Used by the module system */
        TAGGED printname;           /* or sibling pointer | 1 */
                                    /* or parent pointer | 3 */
        struct definition *exported_to; /* Used by module system */
        struct mod_def *imported_from;  /* Used by module system */
        union {
          struct {
            unsigned int spy:1;
            unsigned int breakp:1;
            unsigned int wait:1;          /* obeys declaration */
            unsigned int multifile:1;     /* -"- */
            unsigned int parallel:1;      /* -"- */
            unsigned int dynamic:1;       /* -"- */
            unsigned int nonvar:1; /* seen P(X,...):-var(X),!, */
            unsigned int var:1; /* seen P(X,...):-nonvar(X),!, */
           prop;
         short all;
         properties;
        short predtyp;
        union definfo code;
};
```

In the pre-module system there was a `public` property bit which was used to mark which system predicates to be visible to the user. This property obeyed the public declarations of the source files. There was also a `lock` property bit used to stop these predicates from being redefined. This mechanism has now been replaced by another, as explained below.

3.3 Support Functions

The functions for looking up predicates are supplied with a module pointer instead of, as previously, a predicate table pointer. This enables the functions to initialize the module field if the predicate has to be created.

```
struct definition *find_definition(m,term,argl,insertp)
     struct mod_def *m; /* The module to look up the predicate in */
     TAGGED term,**argl;
     BOOL insertp;
         /* True if the predicate shall be created if not present */

struct definition *parse_definition(complex, m)
    TAGGED complex;
    struct mod_def *m; /* The module to look up the predicate in */
```

The function `parse_definition` is mainly used for loading/compiling and uses the type-in module. There is however times when a different module is used. For example, the `assert` predicates calls the support predicate `interpreted_predicate` which uses `parse_definition`.

```
$interpreted_predicate(+Spec, +Props, +Module)  %Extended with Module
```

A function for looking up modules in the module table is also needed:

```
struct mod_def *find_module(name, insertp)
    TAGGED name;
    BOOL insertp;      /* True if the module shall be
                          created if not present */
```

Some new support predicates are introduced to examine and affect the module system.

```
$typein_module(?OldModule, ?NewModule) %Sets/Reads the type-in module
current_module(?Module)      %Module is defined in the system
```

3.4 The System Module

The *system module* is in many aspects handled similarly to any other module but needs some special treatment. It is created at an early point of the startup sequence and the type-in module is set to the system module. The C-predicates are initialized and the prolog boot file is loaded into the system module.

The file `sys_boot.pl` is then loaded. It contains a handler for module declarations which is used only for the system module, and the handler for meta predicate declarations. When the `sys_module.pl` file, which contains the actual module declaration is loaded, the public predicates list is handed to the C-predicate `make_exportlist` while the meta predicate declaration is handled as for any module.

```
$make_exportlist(+Module, +Publics)
```

This predicate looks up all predicates in the list and links their definition records into a list using the `exported_to` field. This field is not used otherwise for the system predicates. The linked list starts at the `public_lst` field of the system module's module record, which is used only for this purpose.

The point of this is to enable a very fast import of the public system predicates to all other modules as they are created. This is done by the function `import_builtins` which traverses the linked list making all the public predicates visible in the newly created module.

```
void import_builtins(new_modp)     /* Imports public builtins */
    struct mod_def *new_modp;
```

Now the system module is finished and the type-in module is set to the user module which is hereby created. Finally the control is transferred to the top-loop.

3.5 Importation

Import of predicates from a module takes place when the file where that module is defined is loaded. The importing module is the type-in module.

We first give a description of the internal records necessary to keep track of the different files, modules and predicates:

```
$source file(File, Module)      % File is loaded into Module
$source file(Head, File, Mod)   % Predicate with Head is loaded from
                                % File into Module
$curmod(Module, File, Publics)  % Module is defined in File, Publics
                                % is the public predicates
$tmpmod(Module, File, Publics)  % Same as $curmod but only exists
                                % during loading of File
$curimp(ModuleFrom, ModuleTo, Exported) % Used during loading of file
$allimp(ModuleFrom, ModuleTo, Imported) % Current set of imported
                                        % predicates
$loadmode(LoadMode, ImportMode)  % Used during loading of file
```

When the module declaration is processed, a number of things happen: The `$source file/2` record is rerecorded because it was previously recorded with the current type-in module assuming that the file was not a module file. A check is made that the module is not previously defined from another file. If that's the case, the loading

is canceled. It is also checked for attempts to redefine the user module or the prolog module. The `$tmpmod/3` is recorded and the public list is compared to the old public list. If it has been changed, the list is checked to see that all predicates imported by other modules are still public. If not, a warning is issued.

The public list is then processed to determine the set of predicates that is actually to be imported. This depends on the mode of import and if there are predicates previously imported from the same module. There may also be clashes with predicates defined locally or imported from other modules. The set is recorded with the `$curimp/3` record.

All predicates in the module to be loaded are then undefined and the type-in module is set.

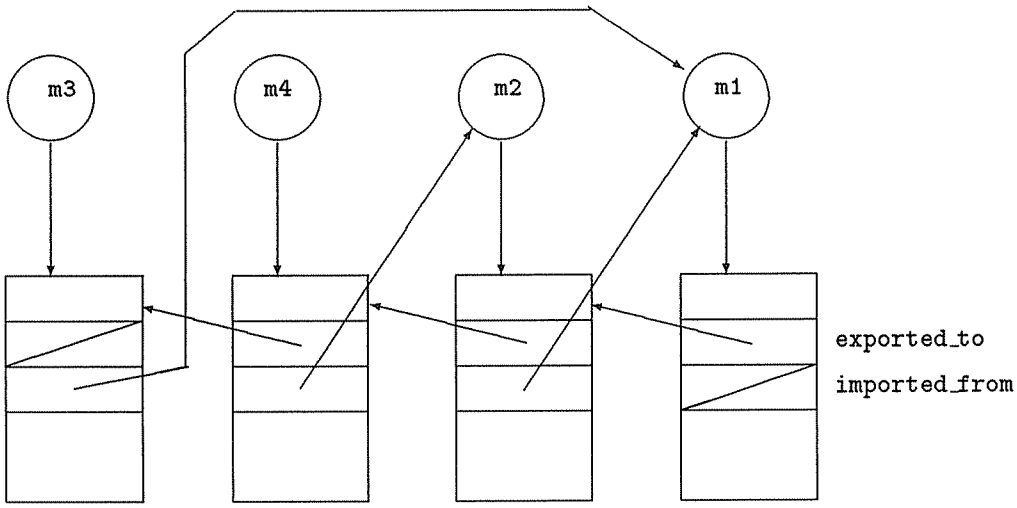
```
$undefine_all(+Module)  % Undefines all predicates in Module
```

After the file is loaded the `$tmpmod/3` record is made into the permanent `$curmod/3`. The actual importation is made by the C-predicate `$import_preds/3` using the set of predicates from `$curimp/3`. The record `$allimp/3` is rerecorded to reflect the set of imported predicates.

```
$import_preds(+Module, +ModuleTo, +Preds)
        % Imports Preds from Module to ModuleTo
```

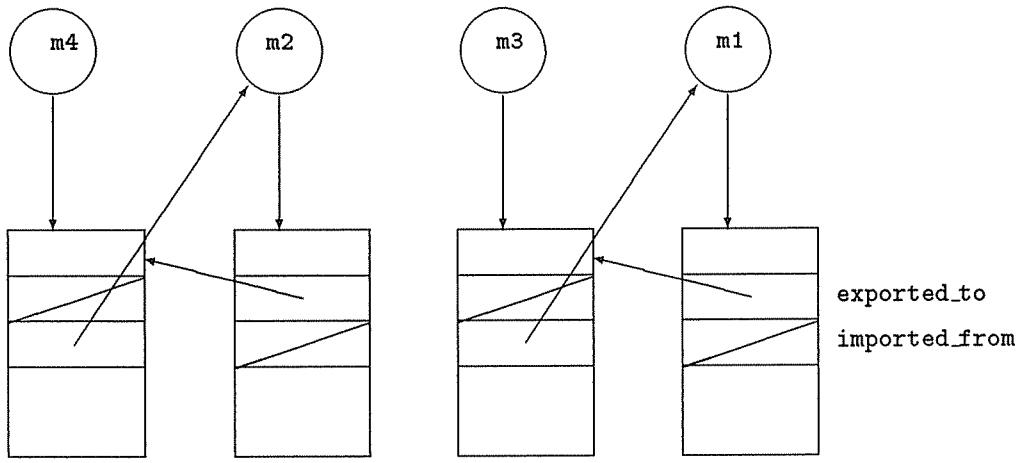
To enable redefinition of an imported predicate, it is not possible to just make the predicate visible in the importing module as is done when importing the system predicates. Instead a local definition record is made as a copy of the original. The `imported_from` field of the copy is set to the module from where the importation is made. The `exported_to` field is used to make a linked list of definition records which starts in the original definition. The following example will clarify the use of the different pointers:

Assume that the predicate `p/1` is defined in module `m1`. It is imported by modules `m2` and `m3`, and reexported by `m2` to `m4`. This will be reflected in the data structures as the figure demonstrates.



Before redefinition

If `p/1` is redefined as a local predicate in `m2`, the `exported_to` field of `p/1` in `m1` is set to `p/1` in `m3`, the `exported_to` field of `p/1` in `m4` is set to nil and the `imported_from` field of `p/1` in `m2` is set to nil.



After redefinition

The scheme employed here enables all kinds of imports, reimports and redefinitions

while demanding a minimum of pointer fields in the definition records. All importing definition records have to be updated whenever a change is made to the origin.

The `module` field in the definition record always points to the module where the predicate is defined. It is used in some of the built in predicates i.e. `spy/1`, `abolish/1-2` and by the debugger.

```
$imported(+Head, +ModuleTo, ?ModuleFrom)
$home_module(+Head, +Module, ?HomeModule)
```

3.6 Meta Declarations

When a meta declaration appears in a source file, an internal record is made for each predicate in the declaration, e.g:

```
:- meta_predicate p(:).
```

gives rise to the internal record

```
$metapred(p(:), m)
```

For file-to-file compilation, another record is used to avoid interference with incore compiled meta predicates.

```
$fmetapred(p(:), m)
```

3.7 Meta Expansion

When compiling, consulting or asserting clauses, all goals that refer to meta predicates must be module name expanded. This is done as follows:

A clause `Head:-Body0` passes through the following piece of code:

```
exp_vars(Head, HV, Module, incore),
wellformed_body(Body0, +, Body, HV, Module, incore)
```

`exp_vars` checks if `Head` is a meta predicate and in that case collects into `HV` all variables that appear in meta expandable argument positions. `wellformed_body` parses the body and, in addition to making sure that the clause is well formed, calls `goal_exp` for each single goal.

```
wellformed_body(Goal, _, Goal1, HV, Module, Mode) :-
        goal_exp(Goal, Goal1, HV, Module, Mode). % meta expansion
```

goal_exp checks if the goal is a meta predicate and performs the module name expansion of the appropriate arguments. The expansion is *not* made if the argument already is of the form m:X or if the argument is a variable appearing in the list HV.

The meta declaration of a predicate is picked up by the predicate

```
meta_pred(+Mode, +Module, +Head, -Decl)
```

which determines the home module of the predicate and calls $metapred(Decl,Module) to get the declaration. The Mode argument may be either incore for incore compilation or ql for file-to-file compilation.

In the ql case, meta_pred looks for either $fmetapred(Decl,Module) or $metapred(Decl,prolog). The latter implies that file-to-file compilation uses the set of built in meta predicates present in the compile-time environment. It is actually the meta predicates of the runtime environment that should be used instead, as they may not be the same. This is part of a general problem for file-to-file compilation i.e. the runtime environment is not known at compile time.

Meta expansion is also done on goals given as directives.

3.8 The Emacs Interface

The loading of code through the emacs interface is handled by a predicate, zap_file/3, which is called from emacs.

```
zap_file(ZapFile, EditorFile, Doing)
```

ZapFile is a temporary file containing the code which is to be loaded and EditorFile is the file from where this code is taken. The relation $source file(File, Module) is used to determine which module to use. If there is no relation for the EditorFile, the type-in module is used. In the latter case the user is prompted to confirm that the loading is to take place.

3.9 Meta Interpreter and Debugger

The meta interpreter, while interpreting a clause, has to keep track of the module where the clause is defined. The entries to the interpreter are extended with a second

argument for the module. The module is then propagated during the interpretation and used when goals in the clause are called.

To be able to call a predicate in any module, a special call predicate, `call_module/2`, is introduced. It does the same job as `call/1` but looks up the predicate in the module supplied by the second argument.

Goals like `module:goal` need some special treatment. You can't just reenter the meta interpreter with `module` as the current module and `goal` as the goal. The reason for this is that `goal` may be a meta predicate and in that case it has to be meta expanded. This is achieved by calling a special meta interpreter for unexpanded goals. It interprets the goal in the same way as the ordinary interpreter except for making meta expansion on the simple goals before calling them.

The debugger is extended to keep track of the module and supply information about it to the user. The home module of imported predicates is picked up on entry to the debugger.

3.10 Built in Predicates

A number of built in predicates are extended to accept a module prefix to some argument. The prefixed argument may be a goal, a predicate specification or a file specification. In any case the argument is parsed by `get_module/3` or `get_module_v/3` to get the module and the unprefixed term.

```
get_module(PrefixedTerm, Term, Module).
get_module_v(PrefixedTerm, Term, Module).
        % Accepts variables as Term and Module
```

If the `PrefixedTerm` is unprefixed, `Module` is unified with the type-in module. This should not normally occur due to the meta expansion mechanism.

For the load predicates, `compile/1` etc, the module spec is used to tell in what module the loaded predicates are going to be defined. When loading module files the predicates are defined in the module which is defined in the file. In that case, importation is made to the module specified in the command. All this is done by simply changing the type-in module during the loading. The type-in module is also used for meta expansion.

`fcompile/1` records the module to be used for meta expansion in a record, `$fcompmod(Module)`. `Module` is set to either the module of the module declaration or for non-module files, the module given as prefix.

The built in predicates that handle dynamic predicates i.e. `assert`, `clause` etc, pass the module to the `$current_clauses/3` which is extended with a module argument. The `assert` predicates also have to perform meta-expansion.

There are some built in predicates which take a goal as an argument and somehow make a call to that goal, like `freeze`, `findall` etc. They need no modification since the call will be transformed to `module:goal`. The `:/2` predicate will in turn call `goal` in `module`.

`predicate_property/2` and `current_predicate/2` are extended to first backtrack through the current modules and then through the predicates in each module. The possible properties from `predicate_property/2` are extended with `imported_from(Module)`, `exported` and `meta_predicate(Spec)`. `imported_from(Module)` uses the C-predicate `$imported` to tell if the predicate is imported and, in that case, from where. `exported` reflects whether the predicate is in the public list of its module. The specification for meta expansion given by a meta declaration is reflected by the property `meta_predicate(Spec)`.

When compiling code, the prefixed goals can be optimized. Instead of always resulting in calls to the `:/2` predicate, it is sometimes possible to make a direct call to the prefixed goal in the prefixing module. This can be done when the module is an atom and the goal is non variable and not a meta predicate. Consider the following goal: `m:p(X)`. First it is made into `call(m:p(X))` to prevent it from being treated as a call to `:/2`. It is then made into a call to `(m:p)/1`. This call is treated as a call to a normal predicate until it reaches the `parse_definition` function. There the predicate `p/1` is looked up into the module `m`. Thus the call is as fast as a call to a local or an imported predicate.

CHAPTER 4

Problems and Future Improvements

The module system is included in the ISP versions 1.5 and following. It will undergo corrections and improvements based on the users experiences. Here follow some problems and suggestions that have come up.

4.1 Prefix of Frozen Goals

When a wait-declared predicate gets suspended, the goal causing the suspension is stored on the heap together with a pointer to the predicates definition record. The built in predicate `frozen/2` returns the frozen goal together with the module which is determined from the definition record.

When a goal is frozen using `freeze/2`, it is in fact the goal `freeze(X,Goal)` which is frozen. Since `freeze` is defined in the prolog module, `frozen/2` tries to prefix the goal with `prolog`. But if `frozen/2` is issued from the user module, the goal is meta expand with `user` and hence can not unify with the `prolog:freeze(X,Goal)`.

An easy solution would be not having `frozen/2` as a meta predicate. This would not maintain full compatibility with non-modular Sicstus though, since the user has to deal explicitly with module prefixes.

4.2 Removable Modules

Suggestions have been made to introduce a possibility to delete a module and reclaim the used memory. This could quite easily be done as all references to predicates in the module are made to local copies of the predicate definition records. However the optimization of prefixed calls causes direct pointers to definition records in other modules and must therefore be abandoned.